Elon Glouberman

March 1, 2018

ID: 904769785

Report for Project 3

**StudentWorld.h / StudentWorld.cpp**

* Class derived from Gameworld.
* Private variables:

    vector<Actor\*> actor\_vector; // actor vector that holds all the Actors in the game

    NachenBlaster\* NB; // the player

    int shipsDestroyed; // counts total ships destroyed to determine if new level

    int aliensOnBoard; // counts total aliens on the board to determine if it should make a new alien coming from the left

* Public functions:
  + Constructor
* virtual int init()
  + initializes the NB private variable
  + creates 30 random stars to start moving across the screen
  + returns continue game
* virtual int move();
  + this is the code that runs for every tick of the game and is key to the animation of it.
  + It deletes everything that is dead or has left the screen from the actor\_vector and calls each of the actors’ “do\_something” method to enable it to react properly to other objects in the game as well as to move and animate itself
* virtual void cleanUp();
  + basic cleaning function that deletes the NB, and the actor vector
  + the destructor for StudentWorld.cpp calls this method as well.
* void setText();
  + creates a text across the top of the screen so that the health points, current level, lives, score, number of cabbages and torpedoes are displayed to the user.
* void setDefaults();
  + the move method will call this method when a player dies or goes to the next level
  + the torpedoes are reset and the health percentage is reset as well
* bool completedLevel() const;
  + determines if the user has killed the appropriate amount of ships in order to proceed to the next level. It utilizes the private variable “Shipsdestroyed” to determine this.
* void makeRandomStar(int x);
  + every time the ship sails across the screen, we need it to look like it is moving. That is what the stars are for. This function generates a new star on the left of the screen in a random position on the y-axis.
* void addObject(Actor\* p);
  + We utilize this function in our Actor.cpp to add new objects and actors to this student world.
  + The function takes the actor\* p and adds it to the actor\_vector
* void addShipDestroyed() {shipsDestroyed++;};
  + The alien actor will utilize this function to tell this studentworld that a ship has been destroyed by either a NachenBlaster or a cabbage/torpedo.
* bool checkCollision(const Actor\* a, const Actor\* b) const;
  + this function utilizes the the Euclidean distance function below to determine whether two objects on the screen have collided with each other. Returns true if the two actors have collided and returns false if they have not.
* double euclidean\_dist(int x1, int y1, int x2, int y2) const;
  + calculates euclidean distance between two sets of x and y coordinates. Returns a double.
* void addNewShip();
  + this function allows us to add a new alien to the left of the screen
  + it calculates how many aliens are on the board (using the private variable AliensOnboard) and by random percentages it generates a new alien player, either a smoregon, smallgon, or snagglegon.
* bool isOnLeft(int x, int y);
  + This function is used by the Alien class to determine whether it has the NB on the left of it and if it should shoot
  + It calculates random number and percentages to determine if it should fire a projectile or not at the NB
* Actor\* getOneCollidingAlien(const Actor\* a) const;
  + This function is utilized by the projectile class as well as the NachenBlaster class to determine if it has collided with an alien in this world. Those classes pass its own objects into it and the function will either return an alien that collides with it or a nullptr if none collide.
* NachenBlaster\* getCollidingPlayer(const Actor\* a) const;
  + If a player collides with the projectile or alien, then we return to those classes a pointer to NB
  + Otherwise, we return a nullptr.
* Void deleteDeadActors();
  + This function is utilized in the end of the move function which deletes all actors that have died during the current tick.
  + It iterates through the actor\_vector and if an actor’s isAlive() is set to false, then it deletes it from the game.
* virtual ~StudentWorld();
  + virtual destructor that calls cleanup and deletes the objects in the game.

**How I tested this:**

For this class, I tested it purely by seeing if the program ran successfully and playing the game myself. I had the console spit out whenever I increased points or if a ship lost points. Additionally, I had the console print out the coordinates of each of the objects they were creating initially just so that I know everything was working correctly.

**Actor.h/Actor.cpp**

**Actor class:**

class Actor: public GraphObject

* my Actor class extends the GraphObject base class.
* It utilizes a constructor initializer list to inherit the id, x,y,depth, direction
* Its private member variables include a bool isAlive to determine whether the object is alive or not, bool collided which determines if the object has been struck by a cabbage or a turnip, bool torpHit which determines if the object has been struck by a torpedo, and a StudentWorld\* m\_gw which is a parameter that enables us to correspond with our StudentWorld class

    StudentWorld\* getGw();

* Accessor -> Returns the Student object from the game world
* I have not made it virtual because I don’t want other actors overriding this method

    virtual void doSomething() = 0;

* This is the primary function that every actor will utilize. This is a pure virtual because I don’t want someone to be able to make an Actor object, and at the same time I want all other derived classes to create a doSomething method which is really important

    virtual ~Actor(){};

* Destructor that is declared virtual (all destructors declared virtual when using polymorphism!)

   void setAlive(bool val)

* Simple mutator function that enables each actor object to determine the alive state of the object

   bool getAlive()

* Accessor function that gets the alive state of the actor

    virtual bool isAlien() { return false; };

* Allows us to determine if the object is an alien. Made this a virtual so that the alien class can return true and override this base class method.

    virtual bool hasBeenHitByCabbageOrTurnip()

* Returns bool collided, which accesses whether the object has been hit by projectile turnip or cabbage

    virtual bool hasBeenHitByTorpedo()

* Returns torpHit which tells us if it has been struck by torpedo

   void setTorpedoHit(bool x)

* Function is because we need to be able to tell the alien and nachenblaster classes that it has been struck by a torpedo, so they can use this method

   void setCabbageHit(bool x) {collided = x;};

* Function is so alien and nachenblaster class can know if it was hit or not by a cabbage or turnip

    virtual void collision(){};

* Collision is going to be a required function for most of the derived objects. It is set to virtual so that the doSomething methods can utilize them in different ways.

    virtual bool isDamageableObject() const {return false;};

* Virtual bool function to determine if object can be damaged (alien or nachenblaster). Made virtual so that those classes can override this function and return false.

**How I tested this:**

For this class, I tried to create a new actor object, which failed because it is abstract. Then, I implemented the derived classes for actor and used breakpoints to make sure that they were being constructed properly through their constructor initializer lists that use this class.

Class **DamageableObject**: public Actor

* Damageable object that extends the actor class. We add a new parameter here called hitPoints, which is set to the private variable called hitP. It is important that all damageable objects have hit points because we need to be able to cause damage to them.

    // How many hit points does this actor have left?

    double hitPoints() const {return hitP;};

* Determines how many hit points this actor has left.

    virtual bool isDamageableObject() const {return true;};

* Overrided function from the actor class to return itself true. This indicates it is a damageableObject.

    // Increase this actor's hit points by amt.

    void increaseHitPoints(double amt) {hitP =amt; };

* We increase the hit points by setting the hitP to the amt in the parameter.

    // This actor suffers an amount of damage caused by being hit by either

    // a ship or a projectile (see constants above).

    virtual void sufferDamage(double amt)

* Every damageable object needs to suffer damage and be able to incur damage on oneself in the case of a collision. Function is made virtual just in case a derived object needs to change the rules a little for how much it is damaged.

Class **NachenBlaster**: public DamageableObject

* NachenBlaster is the player that is playing the game. The parameters that it takes in are the ID, x, y, hitpoints, direction, size and depth.
* The two private variables are ints cabbageInventory as well as flatulanceTorps which count the number of cabbages and torpedos the player currently has in stock. We set the # cabbages initially to 30 and torpedoes to 0.

virtual void doSomething();

* DoSomthing method that takes input from the user (using the spacebar, arrow keys, and some other keys) to determine action.
* Made virtual to override the base class Actor’s doSomething method.
* Simple function that makes the parameter m\_gw() (our student world\*) create a cabbage when space bar is hit, create a torpedo if tab is hit (if inventory has any) and moves left, right, up, and down according to the arrows the user presses.

    int get\_cabbageInt()

* Accesses the amount of cabbages left in the inventory

    void setCabbages(int x)

* Sets the amount of cabbages in the inventory, and makes sure that the limit is 30

    int getTorps()

* Gets amount of torpedoes in inventory

    void setTorps(int x)

* Sets the amount of torpedoes in inventory in the case that the player picks up a torpedo goodie.

**How I tested this:**

For this class, I ran a bunch of tests to determine the cabbage inventories along with the Torpedos. I gave the NachenBlaster an unlimited amount of torpedos so I can see and print out to the console the exact location of them along with the functionality. I also moved the ship all over the screen and pressed all the action keys to see if they were doing the right things.

**Star class**

* The star class extends Actor. Does not add any special parameters.

    virtual void doSomething(); //should move one pixel to the left

* This is the necessary doSomething method required by all derived classes of Actor. This method checks to see if the star has left the screen. If it does, then it marks it as dead. If it has not left the screen, then we move it to the left one pixel.

    virtual ~Star(){};

* Virtual destructor of the star class that we don’t use

**How I tested this:**

For this class, I printed out to the console every coordinate of a new star being created. Then, I changed the probability to 14/15 just to see if the probability was actually working, which it was. I then built it and compared it to the main game to see if things looked off or if it looked like it was doing what it was supposed to be doing.

**Explosion class, extends Actor**

* The explosion is introduced when two objects collide and one of those objects get destroyed. It is derived from the Actor class and does not add any special parameters and private member functions.

    virtual void doSomething()

* If we find that the getAlive() returns false, then we immediately return and not do anything.
* If the size is greater than 3.370, which is 1.5 cubed, then we set it to dead and return
* Otherwise, we increase the size.

**How I tested this:**

For this class, I created different explosion objects in the main and froze each frame to make sure it was exploding properly. Then, I collided a bunch of actors into each other to make sure the explosion would happen there as well.

**Projectile class, that extends Actor**

* This class is abstract and cannot be constructed on its own. It serves the purpose to outline the basic projectile behavior that torpedoes, cabbages, and turnips could easily reuse and redefine functions.
* The only private member function that we have here is called doOtherStuff() which we make here pure virtual to make the Projectile class abstract.
* doOtherStuff will be utilized in the Projectile’s doSomething method so that the other projectiles don’t have to redefine their own doSomething methods.

    virtual void rotate()

* we set the rotation to 20 degrees counter clockwise that only two of the derived classes will utilized. I made this virtual because I want the Torpedo class to override it to return nothing in order to not rotate.

    virtual void collision(){};

* collision class that each of the derived classes will override (this is form the actor class)

    virtual void doSomething();

* checks to see if it is alive (if it isn’t then we return)
* it then calls the collision function, which is specific to each derived class once they override it
* then it calls doOtherStuff which is defined for the derive classes differently so we have different behavior here
* we call rotate()-> this will either rotate or for the torpedo’s overridden rotate function, do nothing.

private:

    virtual void doOtherStuff() = 0;

**Class Cabbage, which extends projectile**

* cabbages are fired by NachenBlasters so they can only inflict damage on aliens. Therefore, it is always going to be moving to the right of the screen and will always have this behavior. In this class, the doSomething method is already defined for all projectiles, so the doOtherStuff() and collision() functions are defined here .

    virtual void collision();

* Here, the cabbage will ask the StudentWorld\* object m\_gw() if it has collided with an alien object, using the function getOneCollidingAlien(this) passing in itself.
* We then set an actor pointer to this function and if it is not nullptr, then we found an alien that we collided with.
* Once this happens, we kill that cabbage object and tell the alien object that it has been hit by a cabbage using the setCabbageHit function which is from the Actor class, so all actors have that function. We can’t direct inflict damage to the alien object because we are dealing with an Actor\* , not an Alien\* since we traverse throught the StudentWorld’s actor\_vector.

private:

    virtual void doOtherStuff()

* This function just moves the cabbage over by 8 pixels to the right

**How I tested this:**

For this class, I ran a bunch of tests to determine if when pressing the space bar, the cabbage would fire. I then froze the frames to make sure that the object was actually spinning counterclockwise while it was being fired. I also made sure to print to the console the object that it collided with in order to ensure that it was return back the correct alien ship or nachenblaster.

**Class Turnip, which extends projectile**

* Turnips are only fired by Aliens so the differentiating functions here for doOtherStuff would mean we move to the left a certain amount. We also redefine collision here to grab the nachenblaster from the student world that has collided with the turnip.

virtual void collision();

* Let’s define a NachenBlaster\* a
* Here, we call the Student world and tell it to get the NachenBlaster object that is in the same area as the turnip (meaning they have collided). If a is not equal to a nullptr that means that we have collided with the nachenblaster.
* We then inflict damage on the nachenblaster by calling its sufferDamage function to lose a certain amount of hitpoints.

private:

    virtual void doOtherStuff();

* This function moves the turnip to left by 6 pixels. Turnips only move to the left so this is all that it will do.

**How I tested this:**

For this class, I ran a bunch of tests to determine if going in the line of site of the alien, the turnip would fire. I then froze the frames to make sure that the object was actually spinning counterclockwise while it was being fired. I also made sure to print to the console the object that it collided with in order to ensure that it was return back the correct nachenblaster.

**Torpedo class, which extends projectile**

* The torpedo class is a bit different than the other projectiles because it can belong to an alien and a NachenBlaster ship. Therefore, I have set a boolian private variable called isNachen which determines if the torpedo belongs to the nachenblaster or the alien ship. This class initially assumes it belongs to the nachenblaster, so it moves directly 14 pixels to the right of it when created. If isNachen is false (as passed through the parameter) then we set isNachen to false and turn the direction around 180 degrees and place it to the left of the alien ship upon creation.

    virtual void collision();

* If we belong to a nachenblaster, then we have to look for a colliding alien.
  + will ask the StudentWorld\* object m\_gw() if it has collided with an alien object, using the function getOneCollidingAlien(this) passing in itself.
  + We then set an actor pointer to this function and if it is not nullptr, then we found an alien that we collided with.
  + Once this happens, we kill that torpedo object and tell the alien object that it has been hit by a torpedo using the setTorpHit function which is from the Actor class, so all actors have that function. We can’t direct inflict damage to the alien object because we are dealing with an Actor\*, not an Alien\* since we traverse through the StudentWorld’s actor\_vector.
* If we belong to an alien, then we have to look for a colliding nachenblaster.
  + Let’s define a NachenBlaster\* a
  + Here, we call the Student world and tell it to get the NachenBlaster object that is in the same area as the torpedo (meaning they have collided). If a is not equal to a nullptr that means that we have collided with the nachenblaster.
  + We then inflict damage on the nachenblaster by calling its sufferDamage function to lose a certain amount of hitpoints.

    bool isNB() {return isNachen;};

* Returns whether it belongs to a nachenblaster or not. Returns true if belongs to nachenblaster false if it belongs to alien.

    virtual void rotate() {return;};

* Function that overrides the initial rotate function as defined in the projectile class. We don’t rotate this object so we return nothing.

private:

    virtual void doOtherStuff();

* This function simply moves the object to the left if it belongs to an alien and to the right if it belongs to a nachenblaster ship. It moves 8 pixels in both situations.

    bool isNachen;

};

**How I tested this:**

For this class, I ran a bunch of tests to determine if when pressing the tab bar, the torpedo would fire. I then froze the frames to make sure that the object was not spinning counterclockwise while it was being fired. I also made sure to print to the console the object that it collided with in order to ensure that it was return back the correct alien ship or nachenblaster.

Goodie class, which extends Actor

* The Goodie class does not define any special private variables, but does have a doOtherStuff function that accepts a nachenblaster\*. Every goodie has very similar doSomething methods, so we define a basic goodie class to take care of not rewriting similar code over and over again.

 virtual void doSomething();

* The doSomething Method first checks if the goodie is alive. If it isn’t we return.
* We then call the collision function, which is defined below.
* If its not dead, then call the collision function agin, as defined below.

 virtual void collision();

* The goodie is always on the lookout for a Nachenblaster object because it can’t collide with any other actor on the board.
  + Let’s define a NachenBlaster\* a
  + Here, we call the Student world and tell it to get the NachenBlaster object that is in the same area as the goodie (meaning they have collided). If a is not equal to a nullptr that means that we have collided with the nachenblaster.
  + If we collided, then we increase the Student world’s score, set ourself to dead and then call the doOtherStuff function which is uniquely defined for each goodie because they all have different effects.

private:

    virtual void doOtherStuff(NachenBlaster\* b)=0;

* We set this function to pure virtual, so the Goodie class is abstract and the functions that derive from it need to redefine this crucial function

};

**Extra Life Goodie which extends the Goodie class**

* Everything in this class is exactly as defined in the goodie class except for the redefinition of the doOtherStuff method in private

private:

    virtual void doOtherStuff(NachenBlaster\* a);

* We increase a life for the NachenBlaster object.

**How I tested this:**

For this class, I created a bunch of these objects in the world so that the Nachenblaster can collide with them. I then made sure to check that the lives were being properly incremented.

**Repair Goodie which extends the Goodie class**

* Everything in this class is exactly as defined in the goodie class except for the redefinition of the doOtherStuff method in private

private:

    virtual void doOtherStuff(NachenBlaster\* a);

* We increase the hit points of the nachenblaster object by 10, also making sure we don’t exceed the 50 hit point max.

**How I tested this:**

For this class, I created a bunch of these objects in the world so that the Nachenblaster can collide with them. I then made sure to check the hitpoints were being incremented properly by printing them out to the console.

**Torpedo Goodie, which extends the Goodie class**

* Everything in this class is exactly as defined in the goodie class except for the redefinition of the doOtherStuff method in private

private:

    virtual void doOtherStuff(NachenBlaster\* a);

* We increase the torpedo inventory of the nachenblaster by 5

**How I tested this:**

For this class, I created a bunch of these objects in the world so that the Nachenblaster can collide with them. I then made sure to check that the torpedo inventory was being properly incremented and then fired the new torpedoes from the nachenblaster.

**Alien class, which extends Damageable object**

* Each of the aliens are different in their own ways, yet they all have similar behaviors. This alien class is defines and implements all the basic behaviors of a Smallgon and the other aliens, so the derived alien classes can utilize the same doSomething method and not rewrite much code, other than what’s different about it.
* The alien introduces a speed variable, and an unsigned int scoreVal. In its constructor, we define the private variables speed, deltaX (by how much each alien should move on the x axis) deltaY (by how much each alien should move on the y axis) and a flight\_plan which is initialized to zero. Both the Smoregon and Smallgon increase the players score value by 250 points when colliding with the player, so we set the incScoreval to 250 (the Snagglegon will override this number to 1000).

void doSomething();

* We first check if the alien is alive or if it has flown off the screen. If it is not, then we return.
* We then call collision() which is defined below.
* We then check if its alive again and if it is then we continue
* Now we call a function called checkFlightPlan() which is shown below.
* Then, we call a function called determineToShoot(), which will determine whether the alien should shoot or not at the nachenblaster
* The alien then moves, by calling its move() function ( it moves by its deltaX and deltaY values)
* We decrement the flightplan
* Call collision again.

    bool isAlien() {return true; };

* Overridden function from the actor class that says that this is an alien

    virtual void collision();

* There are three scenarios in which the alien ship collides with something: 1: if it collides with a cabbage or a turnip, 2: if it collides with a nachenblaster torpedo, and 3: if it collides with a nachenblaster object
* 1:
  + we call hasBeenHitByCabbageOrTurnip that will return true if it has been hit by cabbage or turnip, which the cabbage and turnip classes already should have taken care of already.
  + If this is true, then we sufferDamage on ourselves and decrement hitpoints, or explode if we died.
* 2:
  + We have been hit by a torpedo if the hasBeenHitByTorpedo() function returns true. The torpedo class knows to set this to true if it knows it has collided with it.
  + If this is true, we suffer damage on ourselves and decrement hitpoints or explode if we die.
* 3:
  + if we hit a nachenblaster.
  + We need to check this ourselves, so we use our studentworld object to call the getOneNachenblaster(this) function
    - Let’s define a NachenBlaster\* a
    - Here, we call the Student world and tell it to get the NachenBlaster object that is in the same area as the alien (meaning they have collided). If a is not equal to a nullptr that means that we have collided with the nachenblaster.
    - If we collided, then we increase the Student world’s score, set ourself to dead and then introduce an explosion to the world.
* If alien died, we possibly drop goodie (function defined below).

    void setDeltaY(double dy) {m\_deltaY = dy; };

* Function that sets the private variable m\_deltay by dy

    void setDeltaX(double dx){m\_deltaX = dx;};

* Function that sets m\_deltaX by double dx.

    double getDeltaX() {return m\_deltaX;};

* Function that gets our m\_deltaX

    double getDeltaY() {return m\_deltaY;};

* Function that gets our m\_deltaY

    void setSpeed(double speed) {m\_speed = speed;};

* Function that sets the speed, in the case where we need to change the speed.

    double getSpeed() {return m\_speed;};

* Accesses the speed of the alien

    void setScoreVal(int n) {m\_scoreVal =n;};

* We need to utilize this in the snagglegon class to set the points the player received to 1000 and not the default 250.

    void setIncVal(int n) {incScoreVal =n;};

* Sets the val that we increase by n

    void move() {moveTo(getX()+ m\_deltaX, getY()+ m\_deltaY);};

* We move by deltaX and deltaY.

    virtual void possiblyDropGoodie(){};

* This function is virtual because the smallgon does not utilize this, but the snagglegon and smoregon both need to drop different goodies and perform different actions. We set it to return nothing, so the smallgon doesn’t inherit any behaviors that are not supposed to be inherited.

    virtual bool determineToShoot();

* We call a function from the studentworld object called isOnLeft, which determines whether the Nachenblaster is on the left of the object. This allows us to see if we should fire projectiles at the ship.
* The smallgon and smoregon both have these behaviors.
* This is virtual so that the snagglegon can override it and shoot something else with different probability.
* This returns false if it doesn’t shoot and true if it does.

    virtual void checkFlightPlan();

* If flight plan is 0, we determine random flight plan along with its length.
* Otherwise, we take care of the incident if we collide with the top or bottom of the screen
* This is virtual because snagglegon needs to override this function to always move a certain direction (Down and left).

    void setFlightPlan(int x) {flight\_plan = x; };

* We can set a flight plan length by x

    int getFlightPlan() {return flight\_plan;};

* Returns the length of the flight plan that is currently set.

private:

    double m\_speed;

    unsigned int m\_scoreVal;

    double m\_deltaY;

    double m\_deltaX;

    unsigned int incScoreVal;

    int flight\_plan;

};

**Class Smallgon, which extends Alien class**

* Smallgon inherits almost all behaviors of Alien class, except for a changed ID. The reason I didn’t make all the aliens extend from smallgon was for organizational reasons.

**How I tested this:**

For this class, I created a bunch of these objects in the world so that the Nachenblaster can collide with them. I then checked the probability of the smallgon firing the turnips to make sure that was working correctly. I also made sure to compare it to the real game to make sure that the flight plan and direction of the ship wasn’t unusual. I also fired cabbages at it to make sure it exploded and decremented its hit points properly by printing the hitpoints to the screen.

**Class Smoregon, which extends Alien**

* Smoregon is very similar to Smallgon except for the overriding of two functions as seen below

virtual bool determineToShoot();

* Here we define the smoregon’s unique behavior of speeding up towards the NachenBlaster in certain situations. This requires a different flight plan and change of speed which are both defined here.

virtual void possiblyDropGoodie();

* Once it dies, we drop an extra life goodie if a certain probability comes up.

**How I tested this:**

For this class, I created a bunch of these objects in the world so that the Nachenblaster can collide with them. I then checked the probability of the Smoregon firing the turnips to make sure that was working correctly. I also made sure to compare it to the real game to make sure that the flight plan and direction of the ship wasn’t unusual. I also fired cabbages at it to make sure it exploded and decremented its hit points properly by printing the hitpoints to the screen. Made sure that it would drop a goodie as well as perform its special function of speeding towards the nachenblaster.

**Class Snagglegon, which extends the Alien class**

* Snagglegon reimplements two functions seen below. It also sets IncValue to 1000 so it can give a player that many points if they collide.

    virtual void checkFlightPlan()

* We set the travel direction down and to left all the time, unless we hit the bottom or top of the screen (we go up and to the left if we collide with the bottom)

    virtual bool determineToShoot();

* If the nachenblaster is on the left of the snagglegon, then through a certain probability, it will shoot a torpedo. It will bring the torpedo into the studentworld class

    virtual void possiblyDropGoodie();

* This function happens once a snagglegon has died. It will drop either a repair goodie or a torpedo goodie depending on certain probabilities.

**How I tested this:**

For this class, I created a bunch of these objects in the world so that the Nachenblaster can collide with them. I then checked the probability of the Snagglegon firing the turnips to make sure that was working correctly. I also made sure to compare it to the real game to make sure that the flight plan and direction of the ship wasn’t unusual. I also fired cabbages at it to make sure it exploded and decremented its hit points properly by printing the hitpoints to the screen. Made sure that it would drop a goodie as well as perform its special function of speeding towards the nachenblaster. I made sure that the hitpoints of the nachenblaster would go down correctly after colliding with the snagglegon, along with the fact that the score on the screen incremented correctly.

**List of functionality failed to finish:**

* NA

**List of design and assumptions I made**

* Only assumption for design implementation was for the collision implementations. In this case, I had the Projectiles determining their collision with the NachenBlaster or Alien, and I had the Alien determining its collision with the NachenBlaster. I did not double define these functions by having all classes check for all collisions, because that would have been unnecessary.